CMPT 225: Data Structures & Programming – Unit 05 – Lists

Dr. Jack Thomas

Simon Fraser University

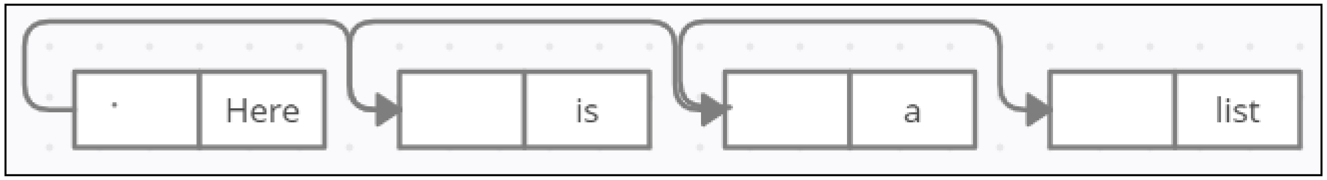
Spring 2021

Today’s Topics

- The List ADT
- Lists in Java
- Singly Linked Lists
- Doubly Linked Lists
- Circular Lists

Lists: What’s The Big Idea?

- An **alternative** data structure to an array, a list organizes data into a series of node objects, each of which includes one data element and a link to one or more other nodes.



- Instead of blocking out a contiguous chunk of memory, you allocate memory dynamically as needed, linking to the next node.

Comparing Arrays and Lists

- **Advantages**:
  - Don’t need a fixed size.
  - Adding or removing from the middle doesn’t involve moving every other element around.
- **Disadvantages**:
  - Harder to access middle elements without an index.
  - No built-in tracking for the length, or where you are in the list.

Anatomy of a List

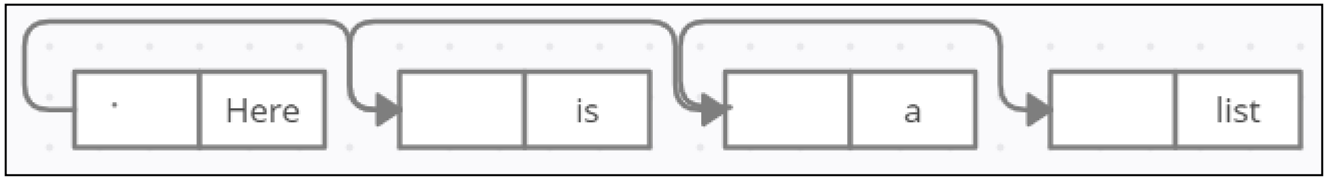
- There's a common set of terminology for lists:
  - A **node** is one entry in the list, an object.
  - The **head** is the node at the top of the list, and is typically kept by whatever object needed a list. There's nothing special built-in to being the head, it's just whatever node you start reading the list from.
  - The **element** is the actual piece of data stored within the node, like one element of an array.
  - One or more **links**, that is, a reference that will take you to the next node in the list (or the previous one, in doubly-linked)
  - The **tail** refers to the last node in the list, indicated by a null link.

Don’t Lists Already Exist?

- Yes, Java already has a List interface, and a number of useful classes that implement it (including a LinkedList and, ironically, ArrayList).
- Our focus today remains on the List concept, as well as learning how these list implementations work on the inside.
- Good to acknowledge you’ll probably mostly use pre-defined lists, though.

The Singly Linked List

- The simplest form of list gives each node one link to the next node.



- Your list keeps track of the head, and possibly the tail, and traverses down the list from link to link (called **link hopping**) whenever it needs to access an element.

Implementing a Singly Linked List

- Will require two different **classes**, one for the **List**, and one for the **Nodes**.
- The **List** keeps track of the head, and should probably also track the size of the list. It will also include methods for adding to, removing from, and searching the list tailored to your program's needs.
- **Nodes** simply need to store their element and a link to the next Node, along with methods for accessing and setting these variables.

```java
class Node {
    private String element;
    private Node next;

    public Node (String s, Node n)
    {
        element = s;
        next = n;
    }

    public String getElement() {
        return element;
    }
    public Node getNext() {
        return next;
    }

    public void setElement(String newElement)
    {
        element = newElement;
    }
    public void setNext(Node newNext)
    {
        next = newNext;
    }
}
```

```java
class SLinkedList {
    protected Node head;
    protected long size;

    public SLinkedList() {
        head = null;
        size = 0;
    }
    /*methods for updating and
    * searching depend on your
    * implementation's goals*/
}
```

Inserting

- Adding to our list is as simple as creating a new node and then setting one of the links in our list to point to it, but where we insert it matters a lot.
- **Adding to the top** of the list is very easy – simply set your new node’s “next” link to point to the current head, then point your list’s “head” link at the new node.
- **Adding to the end** isn’t hard either, so long as you kept a reference to the tail. Just have your new node’s link point to a null, have your current tail point to the new node, and have your list’s “tail” link point to the new node as well.

Insertion Algorithms

```
Algorithm addFirst(v):
        v.setNext(head)
        head <- v
        size <- size + 1
```

```
Algorithm addLast(v):
        v.setNext(null)
        tail.setNext(v)
        tail <- v
        size <- size + 1
```

- Note how we need to remember to **manage our list's size**.
- **Without a tail link**, we'd have to include a step to addLast(v) where we cycle through the whole list from the head.

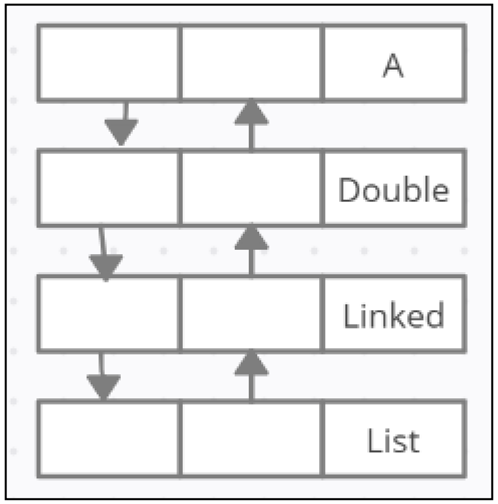
Removing

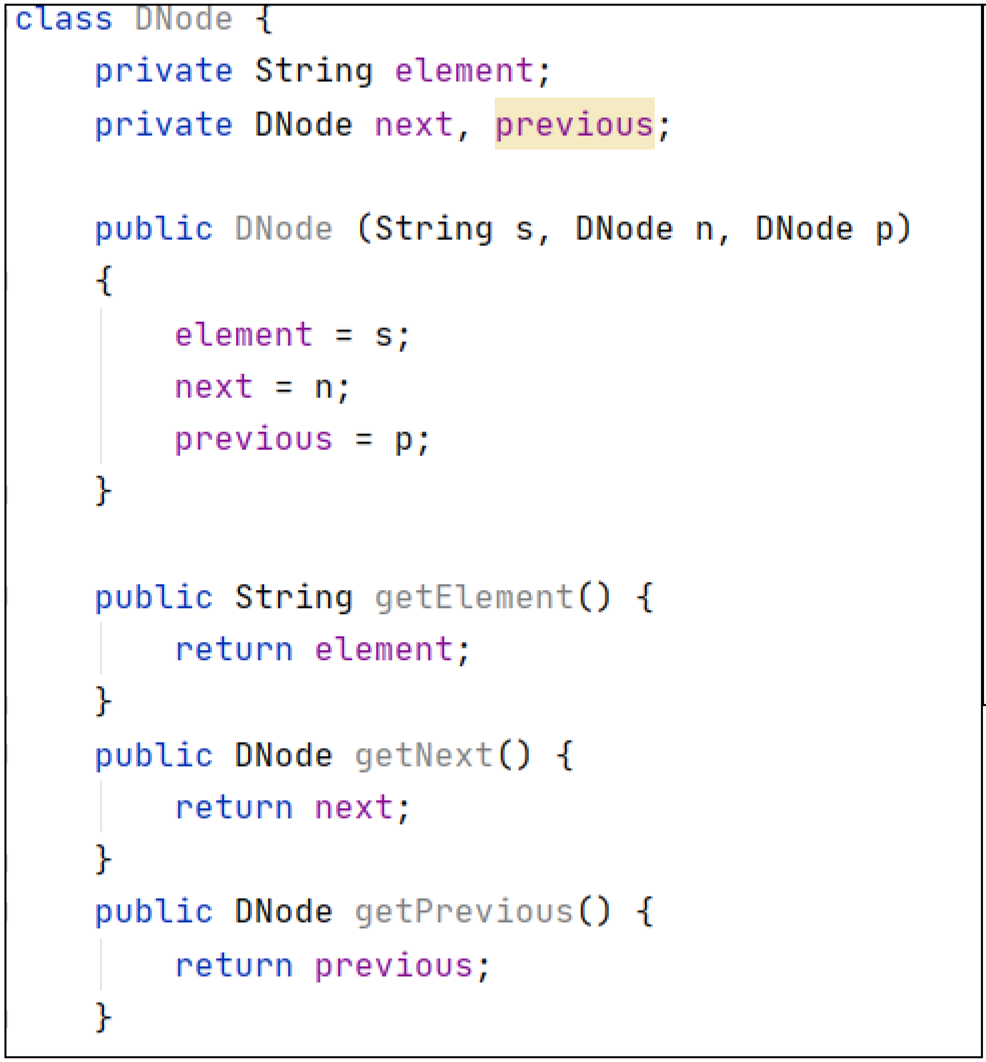
- Removing a head from a list is also fairly simple – just point the list’s head link to the current head’s next link.

```
Algorithm removeFirst():
    if head = null then
        Indicate an error: the list is empty.
    t <- head
    head <- head.getNext()
    t.setNext(null)
    size <- size - 1
```

Doubly Linked Lists

- Removing a tail from a singly-linked list isn't easy, since you'd need to know the node in the list one stop before the tail.
- One solution is to doubly-link each node, so that now they have a link to the next node and the previous node in the list.

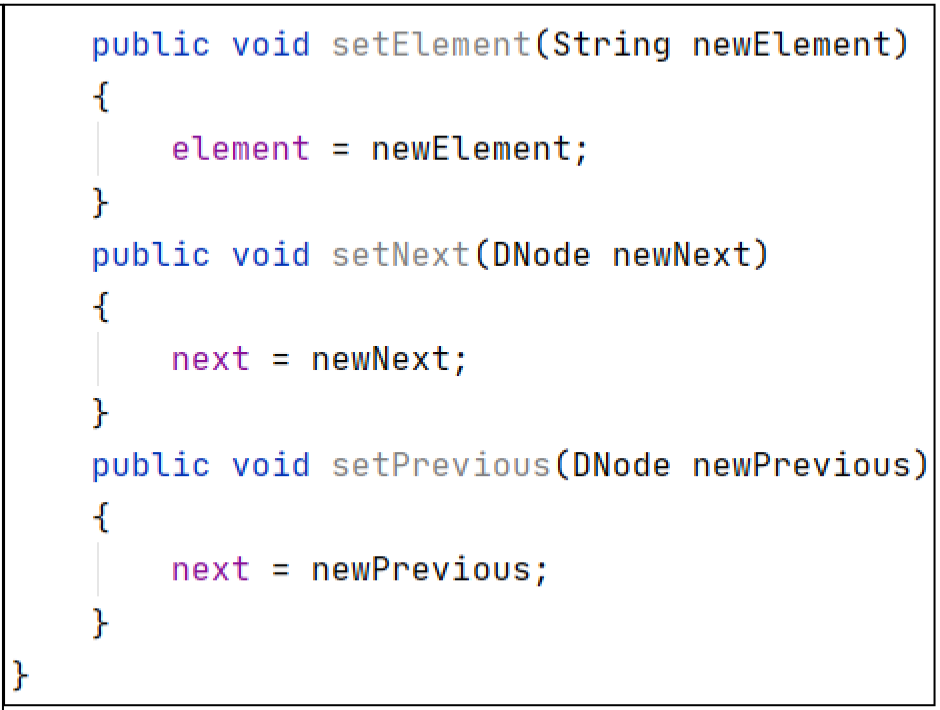




```java
class DNode {
    private String element;
    private DNode next, previous;

    public DNode (String s, DNode n, DNode p)
    {
        element = s;
        next = n;
        previous = p;
    }

    public String getElement() {
        return element;
    }
    public DNode getNext() {
        return next;
    }
    public DNode getPrevious() {
        return previous;
    }
```



```java
    public void setElement(String newElement)
    {
        element = newElement;
    }
    public void setNext(DNode newNext)
    {
        next = newNext;
    }
    public void setPrevious(DNode newPrevious)
    {
        next = newPrevious;
    }
}
```

- The List class will require extensive reworking to take advantage of this relatively small change.

Sentinel Nodes

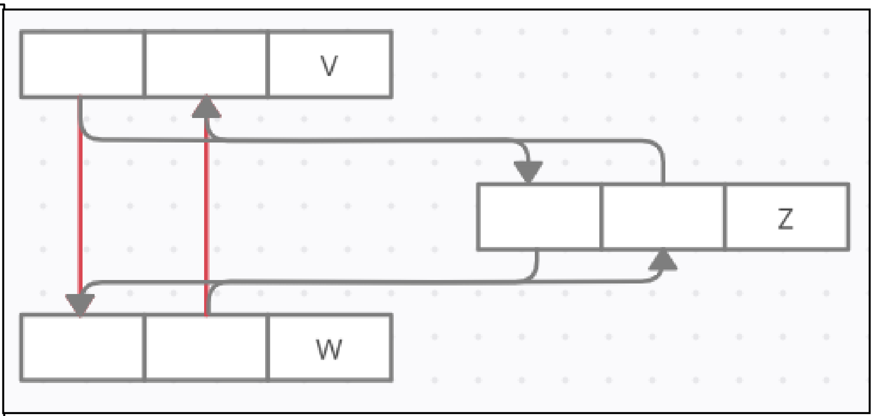
- The head and tail of a doubly-linked list are a little different than the other nodes – the head has a null previous, and the tail has a null next.
- By using dummy **sentinel nodes** with no element inside as the head and tail, we can make some simplifying assumptions in our other functions.
- These dummies are called the **header** and **trailer**, to distinguish them from heads and tails with actual content.

Insertion, or “Linking In”

- The sentinel nodes let us change how we think of adding and removing, since now every regular node has a non-null next and previous.
- Instead of adding to the head or tail, we can imagine adding before a node or after a node.
- This process is called “linking in”, as it’s achieved by connecting the neighbour nodes’ links to the new node.

The “Add After” Algorithm

```
Algorithm addAfter(v,z):
        w <- v.getNext()
        z.setPrev(v)
        z.setNext(w)
        w.setPrev(z)
        v.setNext(z)
        size <- size+1
```



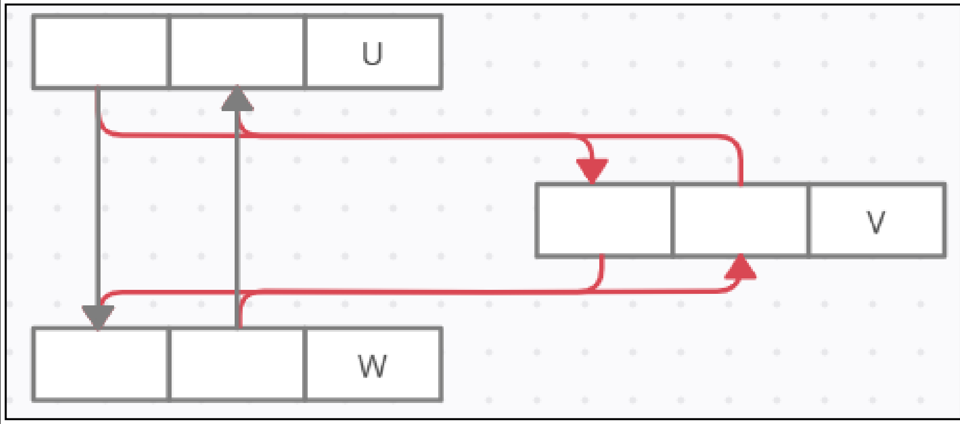
- There’s also “Add Before”, which should follow pretty intuitively from this.

Removal, or “Linking Out”

- The natural inverse of linking in is linking out, when you remove a node by linking its previous and next nodes to each other.
- Don’t forget to **clear the removed node’s next and previous links**, in case it still exists out there somewhere and could cause structural confusion!

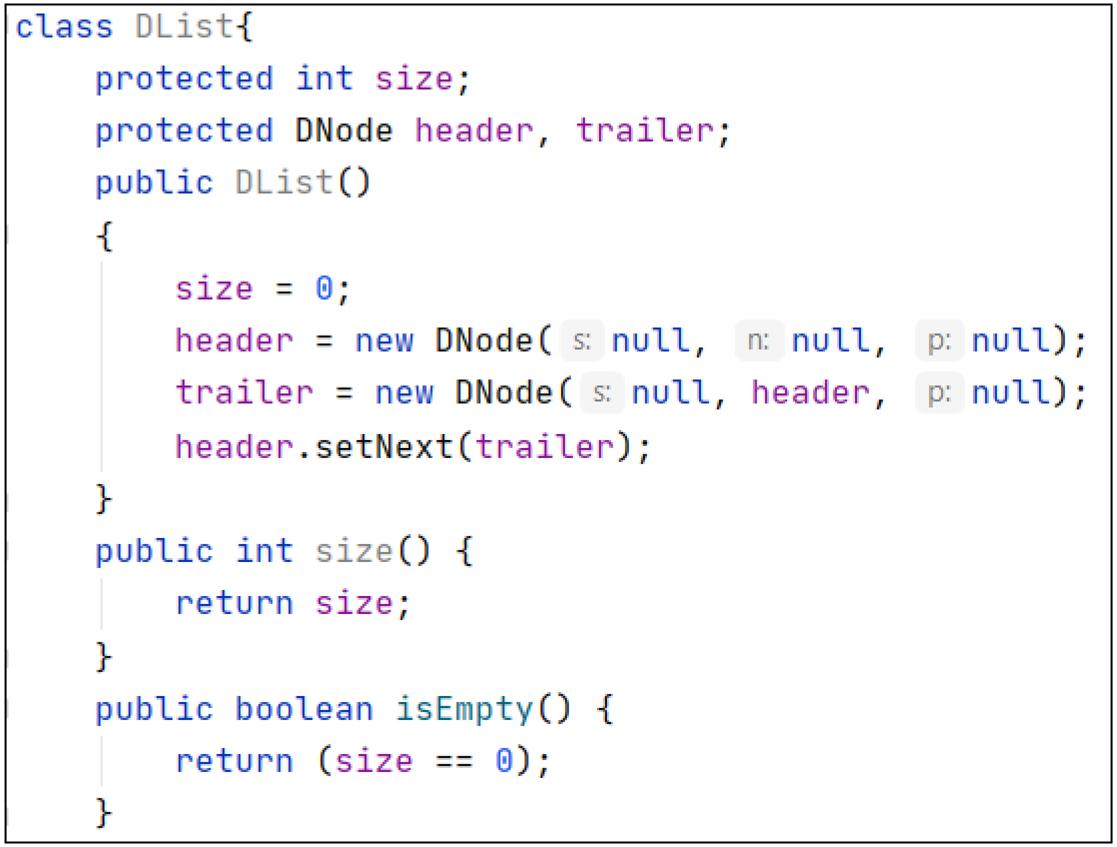
The “Remove” Algorithm

```
Algorithm remove(v):
        u <- v.getPrev()
        w <- v.getNext()
        w.setPrev(u)
        u.setNext(w)
        v.setPrev(null)
        v.setNext(null)
        size <- size - 1
```



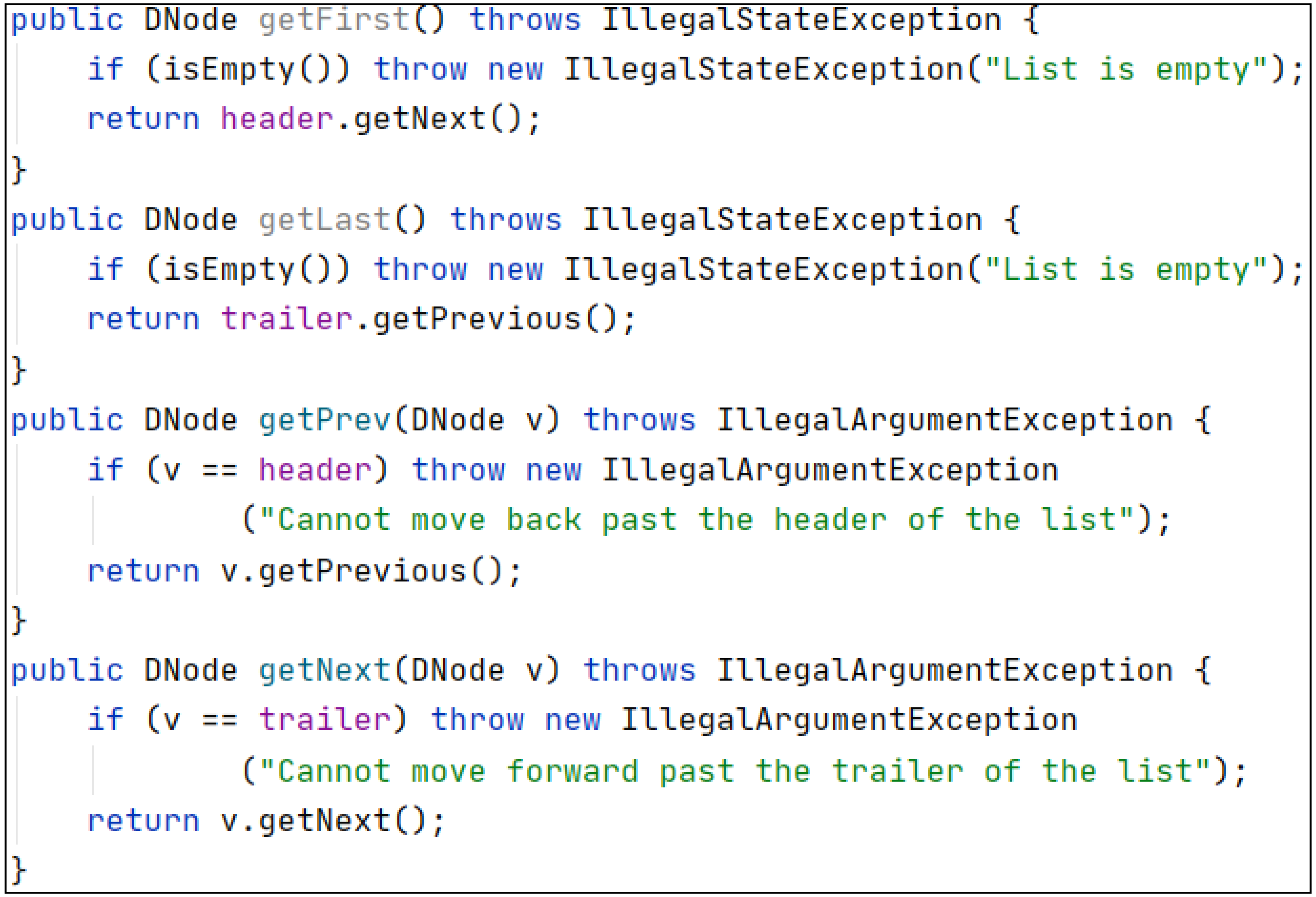
- No two versions this time.

Implementing a Doubly Linked List



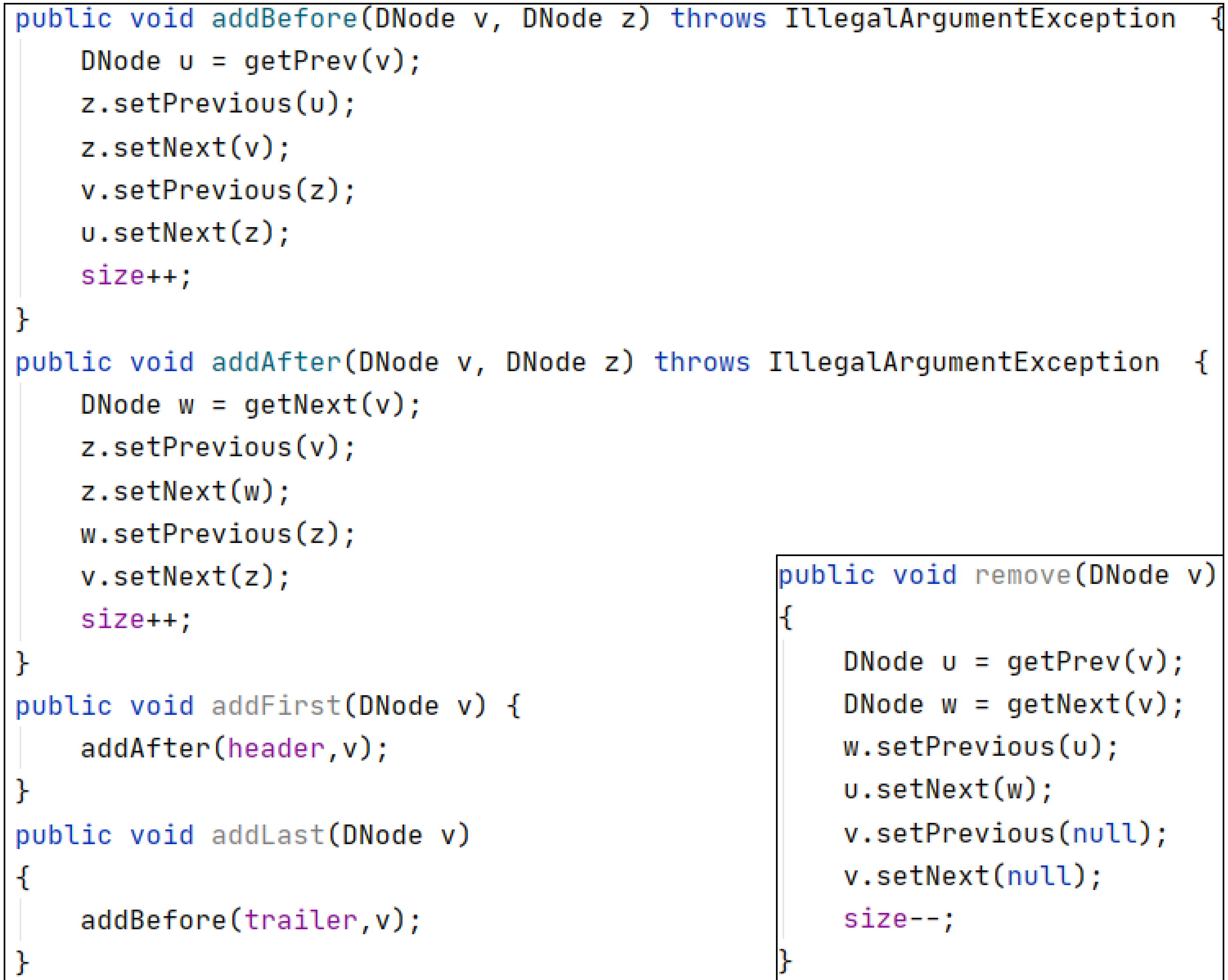
```
class DList{
    protected int size;
    protected DNode header, trailer;
    public DList()
    {
        size = 0;
        header = new DNode( s: null,  n: null,  p: null);
        trailer = new DNode( s: null, header,  p: null);
        header.setNext(trailer);
    }
    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return (size == 0);
    }
```

Implementing a Doubly Linked List

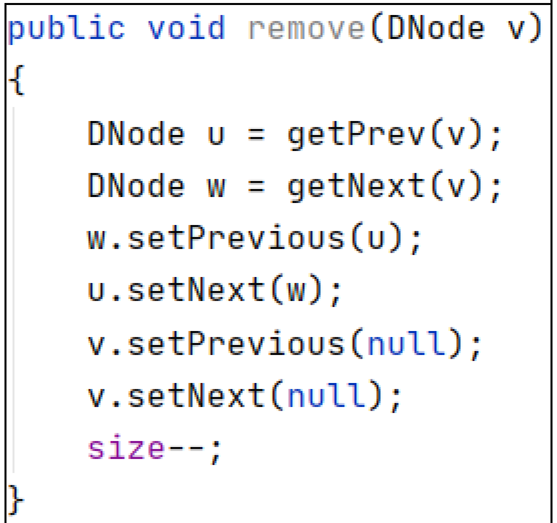


```java
public DNode getFirst() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return header.getNext();
}
public DNode getLast() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return trailer.getPrevious();
}
public DNode getPrev(DNode v) throws IllegalArgumentException {
    if (v == header) throw new IllegalArgumentException
            ("Cannot move back past the header of the list");
    return v.getPrevious();
}
public DNode getNext(DNode v) throws IllegalArgumentException {
    if (v == trailer) throw new IllegalArgumentException
            ("Cannot move forward past the trailer of the list");
    return v.getNext();
}
```

Implementing a Doubly Linked List



```java
public void addBefore(DNode v, DNode z) throws IllegalArgumentException  {
    DNode u = getPrev(v);
    z.setPrevious(u);
    z.setNext(v);
    v.setPrevious(z);
    u.setNext(z);
    size++;
}
public void addAfter(DNode v, DNode z) throws IllegalArgumentException  {
    DNode w = getNext(v);
    z.setPrevious(v);
    z.setNext(w);
    w.setPrevious(z);
    v.setNext(z);
    size++;
}
public void addFirst(DNode v) {
    addAfter(header,v);
}
public void addLast(DNode v)
{
    addBefore(trailer,v);
}
```



```java
public void remove(DNode v)
{
    DNode u = getPrev(v);
    DNode w = getNext(v);
    w.setPrevious(u);
    u.setNext(w);
    v.setPrevious(null);
    v.setNext(null);
    size--;
}
```

Ouroboros: The Circular List

- Exactly what it sounds – link the tail and the head.
- The list now keeps track of a **cursor**, which is, where along the circle we're currently sitting.
- Sort of like a game of "duck, duck, goose".
- Maybe we'll save the full implications of this one for another day!

Recap – The Slideshow Tail

- Lists are an alternative data structure to arrays, that stores each element separately with links to the next (and sometimes previous) element(s).
- Singly-linked lists only link to the next, which can make it tricky to access the middle or end of the list.
- Doubly-linked lists are easier to traverse, and can add things link sentinel nodes to simplify their methods.
- Also sometimes lists go in circles and that’s cool.

[SLIDESHOW TRAILER: DO NOT SHOW]